デバイスの記事

SystemCによるビヘイビア合成入門

第5回 続・ビヘイビア合成可能なC/C++構文

桜井 至

今回は，前回に続いてビヘイビア合成に利用できるC/C++構文について解説する．C/C++をベースとするSystemCを利用すると，設計資産の再利用に有効なオブジェクト指向的な記述を行える．関数のオーバロードやテンプレート，構造体，クラス，継承などの概念を利用して，複雑な設計をコンパクトに表現することができる．（編集部）

前回は，ビヘイビア合成可能な基本的なC/C++構文を紹介しました．今回は，複雑なアルゴリズムの効率的な記述や設計資産の再利用などを容易に行える関数のオーバロード，テンプレート，構造体，クラスなどについて説明します[注1]．

C++は，オブジェクト指向言語と呼ばれています．SystemCはC++に基づいて構築されているため，ハードウェア設計に対してオブジェクト指向の設計スタイルをとることができます．SystemCを用いることで，設計の抽象度を引き上げるだけでなく，アルゴリズム記述，検証，ビヘイビア合成などを含めた設計手法全体について，C++の特徴を生かした手法をとることができます．

リスト1　関数のオーバロードの定義例

```
sc_int<16> average(sc_int<16> a, sc_int<16> b)
{
  return (a+b)/2;
}
sc_int<32> average(sc_int<32> a, sc_int<32> b)
{
  return (a+b)/2;
}
sc_uint<32> average(sc_uint<32> a, sc_uint<32> b)
{
  return (a+b)/2;
}
int average(int a, int b)
{
  return (a+b)/2;
}
```

● 引き数のタイプに合わせて適切な関数を自動呼び出し

前回，関数の簡単な記述例を紹介しました．関数は引き数を取り，計算した結果を返します．関数を用いることで，複数の場所で繰り返し使用する記述を一つにまとめることができます．

例として，与えられた二つの引き数の平均値を計算する，以下のような`average`という関数があるとします．

```
sc_int<16> average(sc_int<16> a,
                   sc_int<16> b)
{
return (a+b)/2;
}
```

ここでは二つの`sc_int<16>`タイプの値の平均値を計算しています．しかし，タイプが`sc_int<16>`ではなく，`sc_int<32>`や`sc_uint<16>`，`int`の場合はどうなるのでしょうか？ C++では，引き数のタイプや個数が異なっていても，**リスト1**のように同じ名まえを付けた関数を複数定義することができます．そのような場合，関数を呼び出す引き数のタイプにしたがって，適切な関数が自動的に呼び出されます．これを関数のオーバロード（function overloading）と言います．

関数のオーバロードによって，同じような処理を同じ関数名で定義して呼び出すことができます．そのため，これらの平均値を計算する関数に対して`average_u16()`，`average_32()`などのまぎらわしい名まえを付ける必要がありません．

注1：本稿で紹介している記述は，米国Forte Design Systems社のビヘイビア合成ツール「Cynthesizer」で合成可能な記述である．そのほかのビヘイビア合成ツールで合成可能かどうかは，それぞれのEDAベンダに確認していただきたい．

ただし，オーバロードの制限として，関数名と引き数のタイプ，および個数が同じで，戻り値のタイプだけが異なる関数をオーバロードすることはできません．これは，コンパイラがどの関数を呼び出すべきか正確に判断できないためです．

なお，関数は，その関数を呼び出す前に関数本体を定義しておかなければなりません．しかし，関数を使用する前に，関数の名まえ，戻り値のデータ・タイプを以下のようにプロトタイプ宣言しておけば，関数の本体をどこの場所に定義していても呼び出すことができます．

```
sc_int<16> average(sc_int<16> a,
                   sc_int<16> b);
sc_int<32> average(sc_int<32> a,
                   sc_int<32> b);
sc_int<32> average(sc_int<32> a,
                   sc_int<32> b);
int average(int a, int b);
```

● **タイプが異なる同じ処理はテンプレートで定義**

関数のオーバロードを利用すると，同じ名まえの関数を定義できます．しかし，定義しているタイプが異なるだけで，まったく同じ処理を行う関数を複数定義する場合はめんどうです．扱うタイプが異なる以外，まったく同じ関数を定義する場合は，テンプレート（template）を使用します．以下にテンプレートの構文を示します．

```
template <テンプレート引き数のリスト>
戻り値のタイプ　　関数名（引き数のリスト）
{
    関数本体
}
```

関数のテンプレートでは，内部で扱うタイプを特定せずに関数を定義します．関数内で使用するタイプは，sc_int<>，intなどの具体的なデータ・タイプを使用せず，Tなどという名まえを指定します．これをテンプレート引き数と言い，<class T>のように記述します．

```
template<class T>
T average(T a, T b)
{
    return (a+b)/2;
}
```

関数テンプレートは，通常の関数の呼び出しと同じように使用します．関数テンプレートを呼び出すと，テンプレート引き数Tを，呼び出しの際に指定したタイプに置き換えた関数が自動的に作成されます．

このように関数テンプレートは，扱うタイプが異なるだけで，そのほかはまったく同じ記述である関数について，記述を一つにまとめることができます．これは関数のオーバロードと似ています．異なる点は，関数のオーバロードの場合，中身の記述が異なっていてもかまわないことです．

● **構造体タイプで複数のデータをパッケージ**

複数の異なるデータを用いる場合，複数のデータをパッケージできる「構造体タイプ」と呼ばれるタイプを宣言できます．構造体タイプは，ユーザ定義タイプの一つです．構造体タイプの宣言には以下のようにstructを用います．

```
struct 構造体タイプ名 {
  タイプ　変数名;
  タイプ　変数名;
   . . .
};
```

いくつかの構造体タイプの宣言例を**リスト2**に示します．RGBタイプはR，G，Bという三つの変数で構成され，それぞれの変数を個別に宣言しています．また，同じように複

リスト2　構造体の宣言例

```
struct RGB
{
  sc_uint<8> R;
  sc_uint<8> G;
  sc_uint<8> B;
};

struct complex
{
  sc_int<16> real;
  sc_int<16> imag;
};

struct pkt
{
  sc_int<8> data;
  sc_int<4> id;
  bool dest0;
  bool dest1;
  bool dest2;
  bool dest3;
 };
```

素数complexは，実数部real，虚数部imagで構成されています．pktはデータとID，およびデスティネーション信号で構成されています．

構造体タイプの変数は，通常の変数と同じように宣言します．例えば，RGBタイプのvar1という変数を宣言する場合には，以下のように記述します．

```
RGB  var1;
```

構造体の宣言時に内部に定義した変数は，メンバ(member)と呼ばれます．メンバ変数にアクセスするためには，ドット演算子(.)を用います．例えば，var1.R = 10; やtmp = var1.Gのように指定して，メンバに値を代入したり，値を参照したりできます．

構造体はSC_MODULE内部の記述で使用するだけでなく，ポートに対して宣言することもできます．複数のデータで構成された構造体タイプを直接ポートに接続してデータの入出力を行えるため，ポート宣言数を減らせます．

● クラスで関数もいっしょにパッケージ

クラス(class)は，変数だけでなく関数もいっしょにパッケージ化できるユーザ定義タイプです．クラス宣言では，以下のように変数や関数をまとめて宣言します．これらの変数や関数は，構造体の場合と同じようにメンバと呼ばれます．

```
class クラス名 {
  アクセス指定子:
  変数の宣言;
```

COLUMN C++を用いたオブジェクト指向設計

C++はオブジェクト指向言語であると言われています．すでにC++を用いてソフトウェアを開発されている技術者の方は，「オブジェクト指向」ということばになじみがあるかもしれません．しかし，HDLを用いてハードウェアを設計している技術者にはなじみのない用語だと思います．SystemCはC++をベースとしているので，ハードウェア設計に対してオブジェクト指向に基づいた手法を適用することができます．

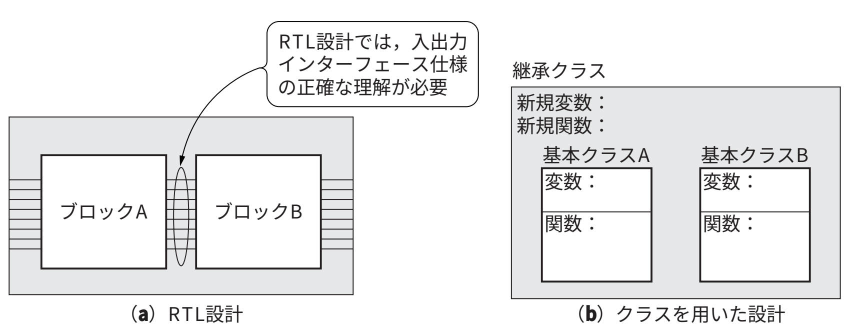
では，オブジェクト指向とはいったいどのようなものなのでしょうか？ 現状のハードウェア設計では，アルゴリズムのフローに基づいて，データの演算処理や変換，制御などを定義していきます．例えば，ディジタル信号処理であれば，データ変換，符号化(圧縮，暗号化)，復号化(伸張)などをフローに沿って記述します．

ハードウェア設計におけるオブジェクト指向では，必要なデータに注目し，それらのデータを処理する関数や演算子をパッケージ化していきます．例えば，画像データ，符号化データ，量子化データなどのそれぞれのデータに注目し，そのデータ処理に必要な関数を定義し，クラス化します．その後，十分に検証された基本クラスを継承することでビルディング・ブロックのように機能を拡張していきます．ハードウェアの処理フローが変わっても，基本となるデータ構造が変化しなければ，基本クラスを再利用しながら機能を拡張していきます．

これはRTL設計におけるボトムアップ設計に似ていますが，RTL設計の場合，設計資産はコンポーネントであるため，接続の際には信号とタイミングの設計が必要です．複雑なハードウェアの場合，正確なタイミング設計を行うためには相応の時間がかかります．また，機能を拡張したり，クロック周期やテクノロジを変更した場合の柔軟性にも問題があります．C++のクラスとビヘイビア合成による設計資産の再利用では，詳細なタイミング設計を行う必要がありません(**図A-1**)．

C++のオブジェクト指向はUML(unified modeling language)に近いものがあります．UMLもオブジェクト指向の設計を可能とします．今後は，UMLをフロントエンドで使用し，それをSystemCに変換した後，ビヘイビア合成を行うことにより，ハードウェアを実現する手法も可能となることでしょう．

現在，ビヘイビア合成ツールの入力言語には，HDL，ANSI C，SystemCなどがあります．SystemCがほかの言語と比べて優位な点の一つは，設計抽象度を引き上げることによってオブジェクト指向の設計手法がとれることと言えます．



図A-1
RTL設計とクラスを用いたビルディング・ブロック設計
RTL設計で既存の資産を再利用するためには，インターフェース仕様を理解する必要がある．クラスを用いたSystemC設計では，クラスを継承することでビルディング・ブロック的な構築が可能となる．ただし，ハードウェア化する場合は，適切なところでモジュール化を行う必要がある．

```
・・・
関数の宣言;
・・・
};
```

このクラス注2を用いることで，実際のデータに着目したオブジェクト指向の設計を行うことができます．このオブジェクト指向については，左掲のコラム「C++を用いたオブジェクト指向設計」を参照してください．

リスト3にクラスを定義した例を示します．これは，前回紹介したFIRフィルタを，クラスを用いて定義したものです．この例では，FIRフィルタの実現に必要なデータ・レジスタを宣言し，さらに係数の読み込みとフィルタ動作を実行する二つの関数を定義しています．このように，クラスを用いることで，対象となるデータ変数とそれに関連した関数をパッケージ化することができます．

クラスを呼び出すことを「オブジェクトを作成する」，「インスタンスを生成する」と言います．オブジェクトの作成は，以下のような記述で行います．

```
クラス名　　クラスの変数名;
```

クラス・タイプの変数は，オブジェクトまたはインスタンスと呼ばれます．オブジェクトの各メンバにアクセスするには，構造体と同じようにドット演算子（ . ）を用います．例えば，**リスト3**のFIRフィルタの関数は**リスト4**のような記述で呼び出します．

1）アクセス指示子

メンバ変数に直接アクセスした場合，不正な値を代入してしまうことがあります．そのようなときに，オブジェクトのメンバのアクセスを制限して，不正な値を代入できないようにします．

メンバがどこからアクセスできるかについて，アクセス指定子を用いて指示します．`private`キーワードを指定すると，クラスの外側からアクセスできなくなります．一方，`public`は，クラスの外側からアクセスできることを指示します．`private`で指示したメンバ変数については，データの正当性をチェックする`public`なメンバ関数を呼び出してからデータを代入することで，不正なデータの代入を防ぎます．

設計資産を作成する人と，実際にその資産を利用する人が異なる場合，不正な値を代入してしまう危険性があります．そのような場合，メンバへのアクセスをチェックすることで，エラーが発生する可能性が減ります．

2）コンストラクタ

オブジェクトの作成と同時に，初期化処理が必要な場合があります．クラスを用いた記述では，このような初期化処理をコンストラクタ（constructor）と呼ばれる特殊なメンバ関数の処理として定義します．コンストラクタは，ク

リスト3　クラスを用いたfir_filterの記述例

```
class fir_filter_t
{
 private:
  sc_uint<8> shift [5] ;
  sc_uint<8> coefs [5] ;
  sc_uint<8> sum_of_coefs;
 public:

  void set_coefs(const sc_uint<8> a [5] ) {
    for(int i=0; i < 5; i++) {
      coefs [i]  = a [i] ;
      sum_of_coefs += a [i] ;
    }
  }

  // フィルタ関数
  sc_uint<8> calc(sc_uint<8> data);

  // コンストラクタ(初期値)
  fir_filter_t() {
    for (int i=0; i < 5; i++) {
      shift [i]  = 0;
      coefs [i]  = 0;
    }
  }
};

sc_uint<8> fir_filter_t::calc(sc_uint<8> data)
{
  sc_uint<16> tmp = 0;

  for (short i=4; i > 0; i--) {
      shift [i]  = shift [i-1] ;
  }
  shift [0]  = data;

  for (short i=0; i<5; i++) {
      tmp += shift [i]  * coefs [i] ;
  }
  return tmp/sum_of_coefs;
}
```

リスト4　fir_filterの呼び出し

```
sc_uint<8> data;
const sc_uint<8> coefs_data [5]  =
    {1,2,3,2,1};
fir_filter_t  var;

var.set_coefs(coefs_data);
data = in_data.read();
result = var.calc(data);
```

注2：`struct`構造体クラスは，変数だけでなく関数もまとめる機能を持つ．そのため，C++ではクラスと構造体の違いはほとんどない．しかし，関数までまとめる場合は，通常，`struct`ではなく`class`を用いる．

ラスからオブジェクトが生成されるときに，自動的に呼び出されます．コンストラクタでは，以下のようにクラス名を関数名として呼び出すようにします．

```
クラス名::クラス名(引き数のリスト)
{
    初期化処理;
}
```

● オブジェクトに対する演算子を新規に定義

intやshortなどのC++データ・タイプ，およびsc_intやsc_uintなどのSystemCデータ・タイプについては，＋演算子や－演算子などを使ってさまざまな演算を行えます．クラスのオブジェクトでも，このような演算子を使えると便利だと思いませんか？

例えば，RGBタイプについて，R，G，Bごとに別々に演算することができますが，次のようにそれらのデータを一度に演算できれば便利です．

```
rgb v1(2,3,5);
rgb v2(10,3,5);
rgb v3 = v1 + v2;
```

C++では，オブジェクトに対する演算子を新しく定義できます．これを演算子のオーバロード(operator overloading)と言います．

演算子のオーバロードは関数の定義に似ています．新

リスト5　RGBクラスの記述例と演算子のオーバロード定義例

```
class RGB
{
 private:
  sc_uint<8> R;
  sc_uint<8> G;
  sc_uint<8> B;

 public:
  // データの書き込み関数
  void writeR(unsigned int a) {
    if (a > 255) R = 255; else R = a;
  }
  void writeG(unsigned int a) {
    if (a > 255) G = 255; else G = a;
  }
  void writeB(unsigned int a) {
    if (a > 255) B = 255; else B = a;
  }
  void write(unsigned int a, unsigned int b, unsigned int c) {
    writeR(a); writeG(b); writeB(c);
  }
  // データの読み出し関数
  sc_uint<8>  readR() { return R; }
  sc_uint<8>  readG() { return G; }
  sc_uint<8>  readB() { return B; }

  RGB read() {
    RGB tmp;
    tmp.R = R; tmp.G = G; tmp.B = B;
    return tmp;
  }

  // ほかのRGBとの平均
  RGB average(RGB a) {
    RGB tmp;
    tmp.R = (R + a.R)/2;
    tmp.G = (G + a.G)/2;
    tmp.B = (B + a.B)/2;
    return tmp;
  }

  // 2値化(スレッショルド値を指定)
  RGB twoValue(int a) {
    RGB tmp;
    if (R > a) tmp.R = 1; else tmp.R = 0;
    if (G > a) tmp.G = 1; else tmp.G = 0;
    if (B > a) tmp.B = 1; else tmp.B = 0;
    return tmp;
  }

  // +加算演算子：RGB同士の加算
  friend RGB operator+(RGB a, RGB b) {
    RGB tmp;
    tmp.R = a.R + b.R;
    tmp.G = a.G + b.G;
    tmp.B = a.B + b.B;
    return tmp;
  }

  // +加算演算子：RGBと定数値の加算
  friend RGB operator+(int a, RGB b) {
    RGB tmp;
    tmp.R = a + b.R;
    tmp.G = a + b.G;
    tmp.B = a + b.B;
    return tmp;
  }

  // 前置インクリメンタ(例：++x)
  RGB operator++() {
    R++; G++; B++;
    return *this;
  }

  // 後置インクリメンタ(例：x++)
  RGB operator++(int d) {
    RGB tmp = *this;
    R++; G++; B++;
    return tmp;
  }

  //論理等価演算
  bool operator==(RGB a){
    return (a.R==R && a.B==B && a.G==G);
  }

  // 値の表示
  void showValue() {
    cout << "R= "<<R<< " G= "<<G<< " B= "<<B<<endl;
  }

  // コンストラクタ定義(初期値0)
  RGB() {
    R = 0; G = 0; B = 0;
  }

  // コンストラクタ定義(宣言時に初期値を代入)
  RGB(int a, int b, int c) {
    writeR(a); writeG(b); writeB(c);
  }
};
```

しい演算子を，一種の関数として定義します．例えば，operator +()という関数を定義することで，＋演算子をオブジェクトに対して使用できます．演算子のオーバロードは，算術演算子や論理演算子，関係演算子などの2項演算子，および単項演算子に対して適用できます．

RGBクラスに対して，一連の関数と演算子のオーバロードを定義した例を**リスト5**に示します．記述例では，さらにR，G，Bそれぞれのメンバ変数に対する値の書き込みと読み出しを行う関数も定義しています．これらの関数を**リスト6**のように記述内で使用できます．

このように，クラスを用いて関数や演算子のオーバロードを定義することで，記述を効率化できます．

● クラスに対してテンプレートを定義可能

関数のテンプレートと同じように，クラスに対してテンプレートを定義して，いろいろなタイプを扱うクラスを作成できます．これらをクラス・テンプレート（class template）と言います．

```
template <テンプレート引き数のリスト>
class  クラス名{
    . . .
};
```

リスト6　オーバロード演算子と関数の使用例

```
RGB x, y, z;     //RGBタイプの変数を宣言
RGB a(10,20,30); //初期値を指定

x.writeR(r_in.read());
x.writeG(g_in.read());
x.writeB(b_in.read());

y.write(1,2,3);
y++;
z = x + y;
z = z.average(a);
z = z.twoValue(127);

r_out = z.readR();
g_out = z.readG();
b_out = z.readB();
```

リスト7　RGBクラスのテンプレート

```
template <class T>
class RGB  {
 private:
  T R;
  T G;
  T B:
 public:
  . . .
```

例えば，前記のRGBクラスに対して**リスト7**のようなテンプレートを定義できます．このテンプレート・クラスを実際に呼び出す際には，以下のように宣言します．

```
RGB<sc_uint<8> > x;
```

また，**リスト3**のFIRフィルタに対して，テンプレート・クラスを指定した記述例を**リスト8**に示します．ここでは，タップ数もパラメータとして渡しています．

SC_MODULEは，class module_name : public sc_moduleのマクロなので，実体はclassです．そのため，SC_MODULEに対してもテンプレートを指定することができます．

RTL設計では，入出力ポートのビット幅や機能をパラメータ化し，設計をインスタンス化する際にそのパラメータを渡すことができました．これによって論理合成時に，別々の設計として具現化しました．同様のパラメタライズド設計は，SystemCではテンプレートを用いて定義します．

リスト8　FIRフィルタ・クラスのテンプレート定義例

```
template <class T, int TAPS>
class fir_filter_t
{
 private:
  T shift [TAPS] ;
  T coefs [TAPS] ;
  T sum_of_coefs;

 public:
  void set_coefs(const T a [TAPS] ) {
    for(int i=0; i <TAPS; i++) {
      coefs [i]  = a [i] ;
      sum_of_coefs += a [i] ;
    }
  }

  T calc(T data);

  fir_filter_t() {
  for (int i=0; i<TAPS; i++) {
      shift [i]  = 0;
      coefs [i]  = 0;
    }
  }
};

//上記 calc関数の本体
template <class T, int TAPS>
T fir_filter_t<T,TAPS>::calc(T data)
{
  int tmp = 0;

  for (short i=TAPS-1; i > 0; i--) {
      shift [i]  = shift [i-1] ;
  }
  shift [0]  = data;

  for (short i=0; i<TAPS; i++) {
      tmp += shift [i]  * coefs [i] ;
  }
  return tmp/sum_of_coefs;
}
```

● 既存のクラスから新しいクラスを作成する継承

すでに作成したクラスから新しいクラスを作成することを「クラスを派生させる」と言います．新しいクラスは派生クラスと呼び，派生したクラスが基本となるクラスのメンバを受け継ぐことを継承（inheritance）と呼びます．また，もとになる既存のクラスを基本クラスと呼びます．派生クラスの宣言は次のように行います．

```
class 派生クラス名:アクセス指示子 基本クラス名{
    追加するメンバの宣言;
};
```

これによって，すでに実績があるクラスの修正や変更を行わずに，新しい機能を追加できます（ビヘイビア合成を行う場合は，ビヘイビア合成可能な構文である必要がある）．例えば，以下のようにRGBクラスとフィルタ・クラスを多重継承させ，そこに新しい機能を追加したimageクラスを定義できます．

```
class image : public RGB,
                        public fir_filter_t {
```

今回は，誌面のつごう上，継承について詳しく紹介していませんが，C++ではこのような継承クラスを用いることでアルゴリズム記述の作成が容易になります．

ビヘイビア合成可能なC++として，テンプレートやクラスなどの構文を中心に紹介しました．今回紹介したように，C++を用いることでオブジェクト指向のハードウェア設計を行うことができます．設計抽象度が上がるだけでなく，アルゴリズム開発や検証について，設計期間の短縮や効率化などの効果を期待できます．

参考・引用*文献

(1) OSCI（Open SystemC Initiative），SystemC Synthesizable Subset，2004.
(2) OSCI，SystemC version 2.0 User's Guide/FUNCTIONAL SPECIFICATION FOR SYSTEMC 2.0，2002.
(3) 長谷川裕恭，小川丈博；SystemCを使ったシステム設計，Design Wave Magazine，2003年7月号～2004年7月号.
(4) Thorsten Grotker，Stan Liao，Grant Martin，Stuart Swan（柿本勝，河原林政道，長谷川隆 監訳）；SystemCによるシステム設計，丸善，2003年.
(5) 大村正之，深山正幸；C/C++によるVLSI設計，共立出版，2003年.

さくらい・たかし
フォルテ・デザイン・システムズ(株)

<筆者プロフィール>
桜井 至．EDA業界に入り，HDLとしてはVerilog HDLから始まり，VHDL，Verilog-AMSへと進み，現在はSystemCを担当．さらに合成ツールとしては，論理合成，アナログ合成と進み，現在はビヘイビア合成を担当．設計言語と合成の大好き人間と言えるかもしれません．著書に「HDL設計入門」，「HDLによるディジタル設計の基礎」，「LSI設計の基礎技術」（テクノプレス社），および「Verilog-AMS入門」（CQ出版社）がある．現在，フォルテ・デザイン・システムズ（http://www.forteds.com/）に所属．

************************ お詫びと訂正 ************************

2005年4月号，特集1第3章「信号処理システム設計を体験する」の記事に誤りがありました．

- p.63，**図9**の説明文の「**COMPLEX TO MAG/PHASE**」は「**MAG/PHASE TO COMPLEX**」の誤りです．
- p.64，式(3)の「$x[n]=A[n]\exp\left(2\pi f_c nT_s+\phi[n]\right)$」は「$x[n]=A[n]\exp\left\{j\left(2\pi f_c nT_s+\phi[n]\right)\right\}$」の誤りです．
  また，**図10**の説明文の「…によってθ_{in}が徐々に…」は「…によってθが徐々に…」の誤りです．
- p.65，**図12**，上段右のループ・フィルタのブロックから右方向へ出ている信号と，左から下段右の＋のブロックに入力されている信号は（垂直なラインで）接続されます．
- p.68，**図16**は，右の画面になります．

お詫びして訂正いたします．

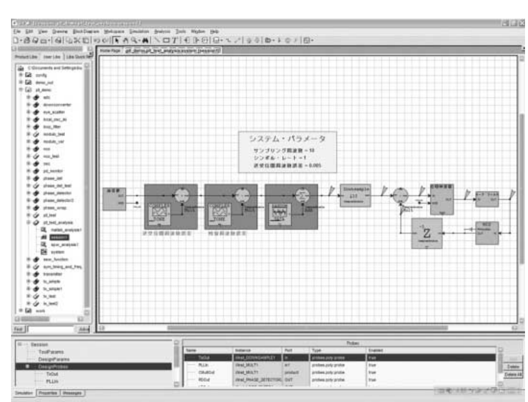


図16　解析用のシステム